Floating Point Simplified RISC Machine Manual
($Revision: 1.14 $)

Gary T. Leavens
Leavens@ucf.edu

November 14, 2023

Abstract

This document defines the machine code of the Floating Point Simplified RISC Machine VM for use (as an example) in the Systems Software class (COP 3402) at UCF.

1 Overview

The Floating Point Simplified RISC Machine (FPSRM) processor's instruction set architecture (ISA) is based on the MIPS processor's ISA [1], but adapted to work with (single-precision) float-point numbers as well as integers. In particular, FPSRM is a little-endian machine with 32-bit (4-byte) words. All instructions are also 32-bits wide and there is no virtual memory support, kernel mode support, or other advanced features.

1.1 Inputs and Outputs

1.1.1 Binary Object Files

The VM is passed a single file name on its command line as an argument; optionally it may also get the option `-p` as an argument. When given a `-p` argument followed by a binary object file name, the VM loads the binary object file and prints the assembly language form of the program, see Section 1.4 details. The remainder of this section is concerned with what the VM does when it only is given a binary object file name on its command line as an argument.

The file name given to the VM must be the name of a (readable) binary object file containing the program that the VM should execute. For example, if the VM's executable is named `vm` and the program it should run is contained in the file named `test.bof` (and both these files are in the current directory), then the VM should execute the program in the file `test.bof` by executing the following command in the Unix shell.

```
./vm test.bof
```

The format of a binary object file (BOF) is given in the header file `bof.h`, which is shown in part in Figure 1. A BOF starts the header, then the instructions (also in binary form) follow, followed by the initial values of data. This layout of binary object files is shown in Figure 2.

The header of a binary object file starts with a 4-character field that contains the characters "`BOF`" (and a null character); this kind of "magic number" is commonly used to identify files in Unix. The magic number is followed (in the BOF header) by the starting address of the program's code and the length of the program's code (in bytes), which constitutes the "text" section of the binary object file. These are followed by the starting address of the data section and its length (in bytes). The data section contains the global/static

variables that the program uses. Finally, the header contains the initial value for the stack and frame pointers, which is the address (in bytes) of the bottom of the runtime stack.

```
/* $Id: bof.h,v 1.14 2023/11/06 09:36:24 leavens Exp $ */
// FLOAT Binary File Format (for the FLOAT SRM)
#ifndef _BOF_H
#define _BOF_H
#include <stdio.h>
#include <stdint.h>
#include "machine_types.h"

#define MAGIC_BUFFER_SIZE 4

typedef struct { // Field magic should be "FBF" (with the null char)
    char     magic[MAGIC_BUFFER_SIZE];
    address_type text_start_address;  // byte address to start running (PC)
    address_type text_length;         // size of the text section in bytes
    address_type data_start_address;  // byte address of static data (GP)
    address_type ints_length;         // size of int data in bytes
    address_type floats_length;       // size of float data in bytes
    address_type stack_bottom_addr;   // byte address of stack "bottom" (FP)
} BOFHeader;

// a type for Binary Output Files
typedef struct {
    FILE *fileptr;
    const char *filename;

// Requires: bf is open for writing in binary
```

Figure 1: The `bof.h` header file that defines the format and operations for binary object files.

1.1.2 Initial/Default Values

The memory of the machine starts at all zero (0) values (i.e., all bits are 0). Then the instructions specified by the given binary object file (as named on the command line) are loaded into memory, starting at address 0, making the contents of the first N bytes (where N is divisible by 4) of memory be the same as the N bytes following the header itself in the binary object file; here N is the same as the header's value of the `text_length` field. Following those N bytes are the bytes of the data section. These are loaded into the memory starting at the data start address given in the header; thus any initial values are copied from the data section of the binary object file into VM's memory.

When the program starts executing:

- the register $gp is set to the start address of the data section given in the header, which must be divisible by 4,
- the registers $fp and $sp are both is set to the stack bottom address given in the header, which must be divisible by 4 and strictly greater than the start address of the data section, and
- the program counter *PC* is set to the text section's start address, which must be divisible by 4 and strictly less than the data section's start address.

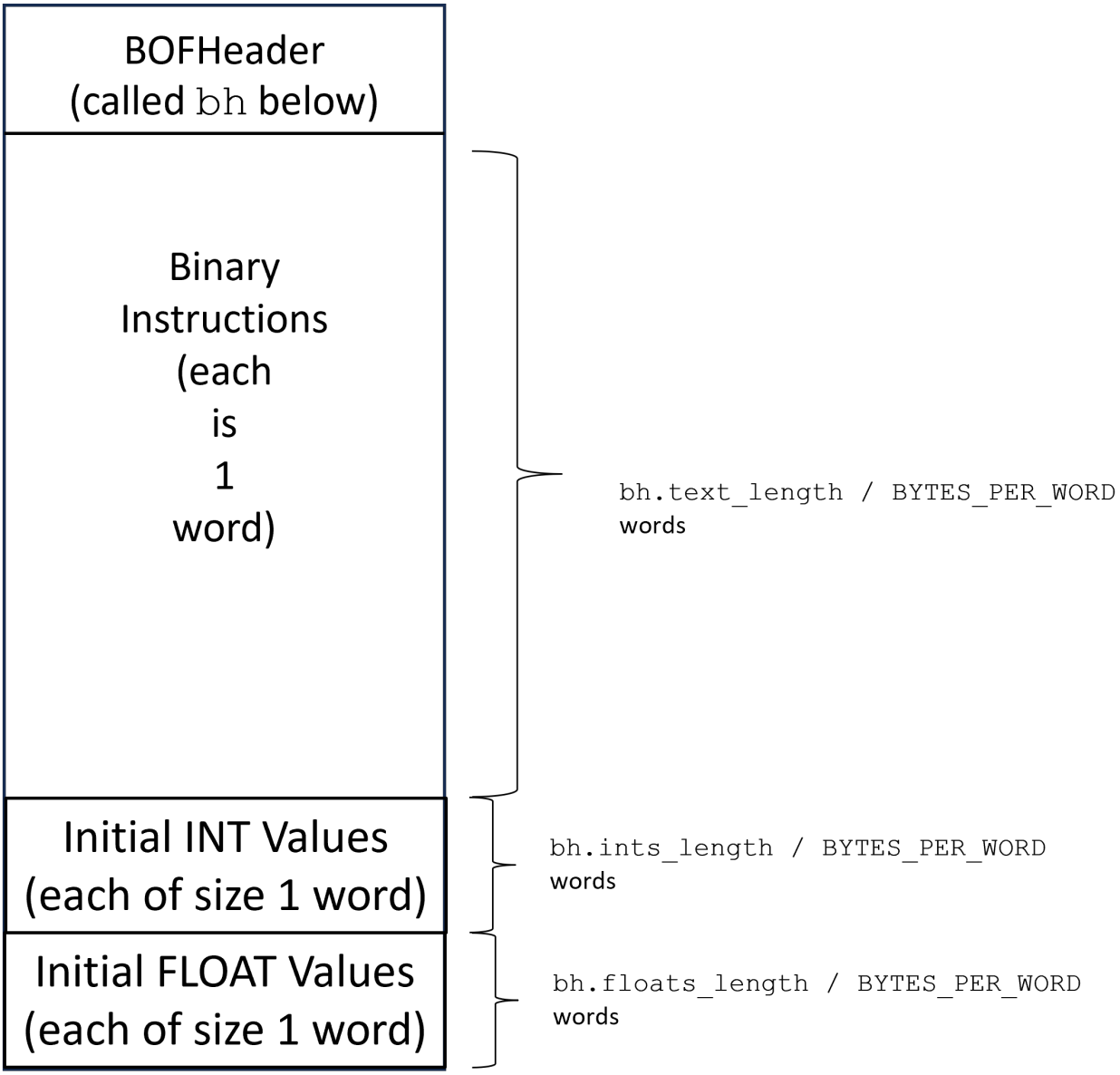

Figure 2: The layout of a binary object file.

```
        # $Id: vm_test0.asm,v 1.1 2023/09/18 03:32:18 leavens Exp $
        .text start
start:  STRA
        ADDI $0, $t0, 1
        EXIT
        .data 1024
        .stack 4096
        .end
```

Figure 3: The FPSRM assembly language file `vm_test0.asm`.

1.2 The Running Program's Input and Output

When the program executes instructions to read or write characters, these are read from standard input (`stdin`) and written to standard output (`stdout`).

However, note that if you want the program to read a character, typing a single character (say `x`) into the terminal (i.e., to the shell) while the program is running will not send that character immediately to the program, as standard input is buffered by default. To send characters to the program it is best to use a pipe or file redirection in the Unix shell; for example, to send the two characters `x` and `y` (followed by a newline character) to the VM running the program `progfile.bof` one could use the following command at the Unix shell:

```
echo xy | ./vm progfile.bof
```

Another command that would accomplish the same thing is to put the characters to be input into a file (using a text editor), say `xy-input.txt` and then to use the following Unix command.

```
./vm progfile.bof < xy-input.txt
```

1.3 Tracing Output

By default, the VM produces output that traces the VM's execution on `stdout`. This tracing output can be turned off by executing a `NOTR` system call instruction and can be turned on by executing a `STRA` system call instruction.[1] (See Section A.4 for more information about these system call instructions.)

The tracing output shows the initial state of the VM, then for each instruction executed, it shows the address (in bytes) of that instruction and then the assembly language form of that instruction.

The state of the machine shown in tracing output shows: the values in the PC, the values of the HI and LO registers (if those are not zero) and the values in all the general purpose registers, then the memory starting at the address in GPR[$gp] with locations containing zeros only indicated with "`...`". and then the data between the addresses between GPR[$sp] and the stack bottom address specified in the binary object file. The idea is that those addresses should include the runtime stack. When showing the memory, locations containing zeros are only indicated with "`...`". For example, when the binary object file that is assembled from the assembly code shown in Figure 3 is executed, which is found in the file `vm_test0.bof`, it produces the output shown in Figure 4.

1.4 Printing the Program

When the VM is given the argument `-p` option followed by a binary object file name, it first loads the instructions and data from the given binary object file, then it prints what was loaded in assembly language

[1] The `STRA` instruction has no effect if the VM is already producing tracing output.

```
      PC: 0
GPR[$0 ]: 0             GPR[$at]: 0             GPR[$v0]: 0             GPR[$v1]: 0
GPR[$a0]: 0             GPR[$a1]: 0             GPR[$a2]: 0             GPR[$a3]: 0
GPR[$t0]: 0             GPR[$t1]: 0             GPR[$t2]: 0             GPR[$t3]: 0
GPR[$t4]: 0             GPR[$t5]: 0             GPR[$t6]: 0             GPR[$t7]: 0
GPR[$s0]: 0             GPR[$s1]: 0             GPR[$s2]: 0             GPR[$s3]: 0
GPR[$s4]: 0             GPR[$s5]: 0             GPR[$s6]: 0             GPR[$s7]: 0
GPR[$t8]: 0             GPR[$t9]: 0             GPR[$k0]: 0             GPR[$k1]: 0
GPR[$gp]: 1024          GPR[$sp]: 4096          GPR[$fp]: 4096          GPR[$ra]: 0
    4096: 0     ...
==> addr:    0 STRA
      PC: 4
GPR[$0 ]: 0             GPR[$at]: 0             GPR[$v0]: 0             GPR[$v1]: 0
GPR[$a0]: 0             GPR[$a1]: 0             GPR[$a2]: 0             GPR[$a3]: 0
GPR[$t0]: 0             GPR[$t1]: 0             GPR[$t2]: 0             GPR[$t3]: 0
GPR[$t4]: 0             GPR[$t5]: 0             GPR[$t6]: 0             GPR[$t7]: 0
GPR[$s0]: 0             GPR[$s1]: 0             GPR[$s2]: 0             GPR[$s3]: 0
GPR[$s4]: 0             GPR[$s5]: 0             GPR[$s6]: 0             GPR[$s7]: 0
GPR[$t8]: 0             GPR[$t9]: 0             GPR[$k0]: 0             GPR[$k1]: 0
GPR[$gp]: 1024          GPR[$sp]: 4096          GPR[$fp]: 4096          GPR[$ra]: 0
    4096: 0     ...
==> addr:    4 ADDI $0, $t0, 1
      PC: 8
GPR[$0 ]: 0             GPR[$at]: 0             GPR[$v0]: 0             GPR[$v1]: 0
GPR[$a0]: 0             GPR[$a1]: 0             GPR[$a2]: 0             GPR[$a3]: 0
GPR[$t0]: 1             GPR[$t1]: 0             GPR[$t2]: 0             GPR[$t3]: 0
GPR[$t4]: 0             GPR[$t5]: 0             GPR[$t6]: 0             GPR[$t7]: 0
GPR[$s0]: 0             GPR[$s1]: 0             GPR[$s2]: 0             GPR[$s3]: 0
GPR[$s4]: 0             GPR[$s5]: 0             GPR[$s6]: 0             GPR[$s7]: 0
GPR[$t8]: 0             GPR[$t9]: 0             GPR[$k0]: 0             GPR[$k1]: 0
GPR[$gp]: 1024          GPR[$sp]: 4096          GPR[$fp]: 4096          GPR[$ra]: 0
    4096: 0     ...
==> addr:    8 EXIT
```

Figure 4: The output of running the VM on file `vm_test0.bof` (which is the result of using the assembler on `vm_test0.asm`), with the command `./vm vm_test0.bof`, as would be printed on standard output.

```
Addr  Instruction
   0 STRA
   4 ADDI $0, $t0, 1
   8 EXIT
```

Figure 5: The output of running `./vm -p vm_test0.bof`. (where the file `vm_test0.bof` is the result of assembling the file `vm_test0.asm` that is shown in Figure 3).

format, and then the VM exits (without running the program). This can be helpful for understanding what a program is doing. This output is shown in Figure 5.

1.5 Error Outputs

All error messages (e.g., for division by zero) are sent to standard error output (`stderr`).

1.6 Exit Code

When the machine halts normally, it exits with a zero error code (which indicates success on Unix). However, when the machine encounters an error it halts and exits with a non-zero exit code (which indicates failure on Unix).

2 Architecture

In FPSRM, words are 32 bits (4 bytes). These bits can be interpreted as a floating-point number, integer, Boolean, or as an address. The machine is byte addressed but instructions must always be at an address that is on a word boundary (i.e., whose address is evenly divisible by 4).

There are separate instructions for manipulating floating-point numbers, which allows an implementation of the FPSRM to track what kind of data is in each word, as that could be useful for tracing.

2.1 Registers

2.1.1 General Purpose Registers and Their Names

The FPSRM is a register machine with 32 general purpose registers[2], numbered 0 to 31 (inclusive). These are all 32-bit registers. Since there are 32 registers, instructions use 5 bits to specify them.

Register 0 cannot be written to, and when read its value is always 0.

Conventions (from the MIPS architecture [1]) are followed for these registers and their names, as shown in Table 1. The names shown in Table 1 are conventional ones.

2.1.2 Special Purpose Registers

FPSRM also has a few special registers. The registers are named:

- *PC*, the program counter which holds the (byte) address of the next instruction to execute,
- *HI*, the high part (i.e., most significant bits) of the result of an integer multiplication or the remainder in an integer division,

[2]What we call "registers" in this document are simply important concepts that simulate what would be registers in a hardware implementation of the virtual machine. For the VM program, these registers would be implemented as variables.

Table 1: FPSRM Register Numbers, Use, and Names

Number	Use	Name
0	always 0 (can't write to this register!)	
1	assembler temporary	$at
$2-3$	function results	$v0, $v1
$4-7$	function arguments	$a0–$a3
$8-15$	temporaries	$t0–$t7
$16-23$	temporaries	$s0–$s7
$24-25$	temporaries	$t8, $t9
$26-27$	reserved for use by OS (don't use!)	
28	globals pointer	$gp
29	stack pointer	$sp
30	frame pointer	$fp
31	return address	$ra

- *LO*, the low part (i.e., least significant bits) of the result of an integer multiplication or the quotient in an integer division.

The *PC* register is manipulated by jump instructions. The *HI* and *LO* registers are read by instructions that move their contents into another register.

The registers that normally hold addresses ($gp, $sp, $fp, $ra) should not have floating-point numbers stored into them and their contents should not be manipulated as floating-point numbers.

2.1.3 Calling Convention

The calling convention on the FPSRM follows the calling convention on the MIPS processor.

That is, the caller must save registers $1-15$, and $24-25$ if they will be needed after a call (and then restores them when needed).

The callee saves (and restores before it returns) registers $16-23$ and $29-31$, if it uses (writes) them.

(Furthermore, register 0 cannot be changed and registers 1 and 28 should not be changed by a hand-written routine. Registers $26-27$ are reserved for the OS should not be changed by user code.)

Note that the jump-and-link (`JAL`) instruction does not save any registers except the *PC*, and it will save that in register 31.

2.2 Binary Instruction Format

In object code, all instructions are one word long and start with a 6-bit opcode. However, instructions may have one of several formats, with the format depending on the opcode (called "op" below). The fields of each instruction format shown in Table 2 are followed by their width in bits; for example the op field is 6 bits wide.

The list of instructions and details on their execution appears in Appendix A.

2.3 Machine Cycles

The FPSRM instruction cycle conceptually does the following for each instruction:

Table 2: FPSRM Instruction Formats

- Register/computational type instruction format:

op:6	rs:5	rt:5	rd:5	shift:5	func:6

- System call instructions, whose format is a variant of the register type instruction format, but with a func field value of 12:

op:6	code:20	func:6

- Immediate operand type instruction format:

op:6	rs:5	rt:5	immed:16

- Jump type instruction format:

op:6	addr:26

1. Let *IR* be the instruction at the location that *PC* indicates. (Note that *IR* could be considered to be the contents of a register.)
2. The *PC* is made to point to the next instruction (i.e., it is set to the address $PC + 4$).
3. Then the instruction in *IR* is executed. The op component of this instruction (*IR*.op) indicates the operation to be executed. For example, if *IR*.op encodes the instruction JR, then the machine jumps to address given the register argument, by setting the *PC* register (to the contents of the given register).

2.4 Size Limits

The following constant (found in `machine.h`) defines the size of the memory for the VM.

```
#define MEMORY_SIZE_IN_BYTES (65536 - BYTES_PER_WORD)
```

Note that `BYTES_PER_WORD` is defined to be 4.

2.5 Invariants

The VM enforces the following invariants and will halt with an error message (written to stderr) if one of them is violated. In these invariants the registers are all treated as holding (byte) addrresses.

- *PC* % `BYTES_PER_WORD` $= 0$,
- GPR[$gp] % `BYTES_PER_WORD` $= 0$,
- GPR[$sp] % `BYTES_PER_WORD` $= 0$,
- GPR[$fp] % `BYTES_PER_WORD` $= 0$,
- $0 \leq$ GPR[$gp],
- GPR[$gp] $<$ GPR[$sp],
- GPR[$sp] $\leq$ GPR[$fp],
- GPR[$fp] $<$ `MEMORY_SIZE_IN_BYTES`,

- $0 \leq PC$,
- $PC <$ `MEMORY_SIZE_IN_BYTES`, and
- GPR$[0] = 0$.

A Appendix A

In the following tables, italicized names (such as s) are meta-variables that refer to integers. If an instruction's field is notated as $-$, then its value does not matter (we use 0 as a placeholder for such values in examples).

All numbers appearing in the following table are in decimal (base 10) notation.

A.1 Register/Computational Instructions

Note that all of the instructions in table Table 3 have an opcode of 0, and a function specified by the func field. They each also have 3 register arguments: rs, rt, and rd. The contents of the general purpose register r is notated as GPR$[r]$ in the table. All numbers in the table are in decimal notation.

Floating-point arithmetic operations are performed as for C **float** values. Integer arithmetic, bitwise, and logical operations are performed as for C **int** values. However, the right shift works on the contents of the register GPR$[t]$ in a logical manner, as if it were an **unsigned int**, so it should shift in zeros from the left.

Table 3: Register Format Instructions

Name	op	rs	rt	rd	shift	func	Comment (Explanation)
ADD	0	s	t	d	-	33	Integer Add: GPR[d] $\leftarrow$ GPR[s] + GPR[t]
FADD	0	s	t	d	-	49	Floating-point Add: GPR[d] $\leftarrow$ GPR[s] + GPR[t]
SUB	0	s	t	d	-	35	Integer Subtract: GPR[d] $\leftarrow$ GPR[s] $-$ GPR[t]
FSUB	0	s	t	d	-	51	Floating-point Subtract: GPR[d] $\leftarrow$ GPR[s] $-$ GPR[t]
MUL	0	s	t	-	-	25	Integer Multiply: Multiply GPR[s] and GPR[t], putting the least significant bits in *LO* and the most significant bits in *HI*. (HI, LO) $\leftarrow$ GPR[s] $\times$ GPR[t]
FMUL	0	s	t	d	-	41	Floating-point Multiply: GPR[d] $\leftarrow$ GPR[s] $\times$ GPR[t]
DIV	0	s	t	-	-	27	Integer Divide (remainder in *HI*, quotient in *LO*): *HI* $\leftarrow$ GPR[s] % GPR[t] *LO* $\leftarrow$ GPR[s]/GPR[t])
FDIV	0	s	t	d	-	43	Floating-point Divide: GPR[d] $\leftarrow$ GPR[s]/GPR[t]
MFHI	0	-	-	d	-	16	Move from HI: GPR[d] $\leftarrow$ *HI*
MFLO	0	-	-	d	-	18	Move from LO: GPR[d] $\leftarrow$ *LO*
AND	0	s	t	d	-	36	Bitwise And: GPR[d] $\leftarrow$ GPR[s] $\wedge$ GPR[t]
BOR	0	s	t	d	-	37	Bitwise Or: GPR[d] $\leftarrow$ GPR[s] $\vee$ GPR[t]
NOR	0	s	t	d	-	39	Bitwise Not-Or: GPR[d] $\leftarrow \neg$(GPR[s] $\vee$ GPR[t])
XOR	0	s	t	d	-	38	Bitwise Exclusive-Or: GPR[d] $\leftarrow$ GPR[s] xor GPR[t]
SLL	0	-	t	d	h	0	Shift Left Logical: GPR[d] $\leftarrow$ GPR[t] « h
SRL	0	-	t	d	h	3	Shift Right Logical: GPR[d] $\leftarrow$ GPR[t] » h
CVT	0	-	t	d	-	4	Convert to Float: GPR[d] $\leftarrow$ `(float)` GPR[t]
RND	0	-	t	d	-	5	Round to Integer: GPR[d] $\leftarrow$`round(`GPR[t]`)`
JR	0	s	0	0	0	8	Jump Register: *PC* $\leftarrow$ GPR[s]
SYSCALL	0	-	-	-	-	12	System Call: (see Table 6)

A.2 Immediate Type Instructions

The instructions in Table 4 may have up to 2 register arguments, and all have an immediate operand, which is a 16 bit integer value.

For arithmetic operations, the immediate value is sign-extended (to a 32-bit **int** value), which is written in the explanations using the function "sgnExt." For example, suppose i is -1, which is FFFF in hexadecimal notation; then sgnExt(i) is FFFFFFFF in hexadecimal, which still represents -1.

However, for logical operations, the immediate value is zero-extended, which is written in the explanations using the function "zeroExt." For example, suppose i is -1, which is FFFF in hexadecimal notation; then zeroExt(i) is 0000FFFF in hexadecimal notation.

The branch instructions BEQ, BGEZ, BGTZ, BLEZ, BLTZ, and BNEQ all treat their register argument as if it were an integer value. However, the branch instructions BFEQ, BFGEZ, BFGTZ, BFLEZ, BFLTZ, and BFNEQ all treat their register argument as if it were a floating-point value.

For the branch instructions, the immediate value, o is first shifted left 2 bits (multiplied by 4) and then sign-extended, which is written as "formOffset" in the table. (Thus formOffset(o) = sgnExt$(4 \times o)$.) Note that the resulting address is added to the address of the instruction following the currently executing instruction, not the address of the instruction itself, since the *PC* has already been advanced.

For loads and stores, memory$[a]$ denotes the contents of the machine's memory at the byte address a.

A.3 Jump Type Instructions

The instructions in Table 5 have a 26-bit field "addr" which is used to form the address to jump to. Forming this address is done by left-shifting the given "addr" field, a, by 2 bits, and then concatenating the (4) high bits of the *PC* with those $26 + 2$ bits to form a 32-bit address. This is written "formAddress$(PC, a,)$" in the table. For example if a is DECADE in hexadecimal notation, and *PC* is FFFACADE in hexadecimal notation, then formAddress(PC, a) is F37B2B78 in hexadecimal notation. (Note: if the high-order 4 bits of *PC* are 0, then formAddress(PC, a) is equivalent to left-shifting a by 2 bits.)

The Jump and Link (JAL) instruction does a subroutine call. It does not explicitly manipulate the runtime stack.

A.4 System Calls

System calls are used to provide operating system services. System calls are made by instructions with op 0 and func 12 having the following format (with code field made of the 20 bits of what would be the rs, rt, rd, and shift fields of a register type instruction, all combined). The code field is used to specify the service requested.

System calls include exiting a program and various kinds of printing and reading of character data (bytes). These are described in Table 6, using C library equivalents. In the table, an entry of $-$ means that the contents of argument registers is not specified. Otherwise, the contents of particular argument registers are used to pass actual arguments to the system calls (the program must load the actual argument values into those registers before making the call). Only the PFLT system call treats its argument (in $a0) as a floating-point number. (Recall that the correspondence between named registers and register numbers is given in Table 1.) All printing done by these instructions is to the VM's standard output (stdout) and reading is from the VM's standard input (stdin).

Implementing the PSTR instruction is a bit tricky, because the `printf` function from the C standard library function will expect a C pointer to characters as its argument, but that can be given to it by giving it the address of those character's representations in the memory starting at the VM address given by the contents of GPR[$a0]. A similar trick can be used for RFLT, but the code snippet given in Table 6 will also work.

Table 4: Immediate format instructions:

Name	op	rs	rt	immed	Comment (Explanation)
ADDI	9	s	t	i	Add immediate: GPR[t] $\leftarrow$ GPR[s] + sgnExt(i)
ANDI	12	s	t	i	Bitwise And immediate: GPR[t] $\leftarrow$ GPR[s] $\wedge$ zeroExt(i)
BORI	13	s	t	i	Bitwise Or immediate: GPR[t] $\leftarrow$ GPR[s] $\vee$ zeroExt(i)
XORI	14	s	t	i	Bitwise Xor immediate: GPR[t] $\leftarrow$ GPR[s] xor zeroExt(i)
BEQ	4	s	t	o	Branch on Equal: **if** GPR[s] = GPR[t] **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BGEZ	1	s	1	o	Branch ≥ 0: **if** GPR[s] ≥ 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BGTZ	7	s	0	o	Branch > 0: **if** GPR[s] > 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BLEZ	6	s	0	o	Branch ≤ 0: **if** GPR[s] ≤ 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BLTZ	8	s	0	o	Branch < 0: **if** GPR[s] < 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BNE	5	s	t	o	Branch Not Equal: **if** GPR[s] $\neq$ GPR[t] **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BFEQ	20	s	t	o	Branch Float Equal: **if** GPR[s] = GPR[t] **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BFGEZ	17	s	1	o	Branch Float ≥ 0: **if** GPR[s] ≥ 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BFGTZ	23	s	0	o	Branch Float > 0: **if** GPR[s] > 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BFLEZ	22	s	0	o	Branch Float ≤ 0: **if** GPR[s] ≤ 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BFLTZ	24	s	0	o	Branch Float < 0: **if** GPR[s] < 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BFNE	21	s	t	o	Branch Float Not Equal: **if** GPR[s] $\neq$ GPR[t] **then** *PC* $\leftarrow$ *PC* + formOffset(o)
LBU	36	b	t	o	Load Byte Unsigned: GPR[t] $\leftarrow$ zeroExt(memory[GPR[b] + formOffset(o)])
LW	35	b	t	o	Load Word (4 bytes): GPR[t] $\leftarrow$ memory[GPR[b] + formOffset(o)]
FLW	41	b	t	o	Load Word Float: GPR[t] $\leftarrow$ memory[GPR[b] + formOffset(o)]
SB	40	b	t	o	Store Byte (least significant byte of GPR[t]): memory[GPR[b] + formOffset(o)] $\leftarrow$ GPR[t]
SW	43	b	t	o	Store Word (4 bytes): memory[GPR[b] + formOffset(o)] $\leftarrow$ GPR[t]
FSW	42	b	t	o	Store Word Float: memory[GPR[b] + formOffset(o)] $\leftarrow$ GPR[t]

Table 5: Jump Type Instructions

Name	op	addr	Comment (Explanation)
JMP	2	a	Jump: *PC* $\leftarrow$ formAddress(PC, a)
JAL	3	a	Jump and Link: GPR[$ra] $\leftarrow$ *PC*; *PC* $\leftarrow$ formAddress(*PC*, a)

Table 6: System Calls

code	name	arg. reg.	Effect (in terms of C std. library)
10	EXIT	-	`exit(0) // halt`
4	PSTR	$a0	GPR[$v0] ← `printf("%s",&`memory[GPR[$a0]]`)`
5	PINT	$a0	GPR[$v0] ← `printf("%d",`GPR[$a0]`)`
6	PFLT	$a0	GPR[$v0] ← `printf("%f",`GPR[$a0]`)`
11	PCH	$a0	GPR[$v0] ←`fputc(`GPR[$a0]`,stdout)`
12	RCH	-	GPR[$v0] ← `getc(stdin)`
13	RFLT	-	**`float`** `f; fscanf(stdin,"%f",&f);` GPR[$v0] ← `f`
256	STRA	-	start VM tracing; start tracing output
257	NOTR	-	no VM tracing; stop the tracing output

References

[1] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.